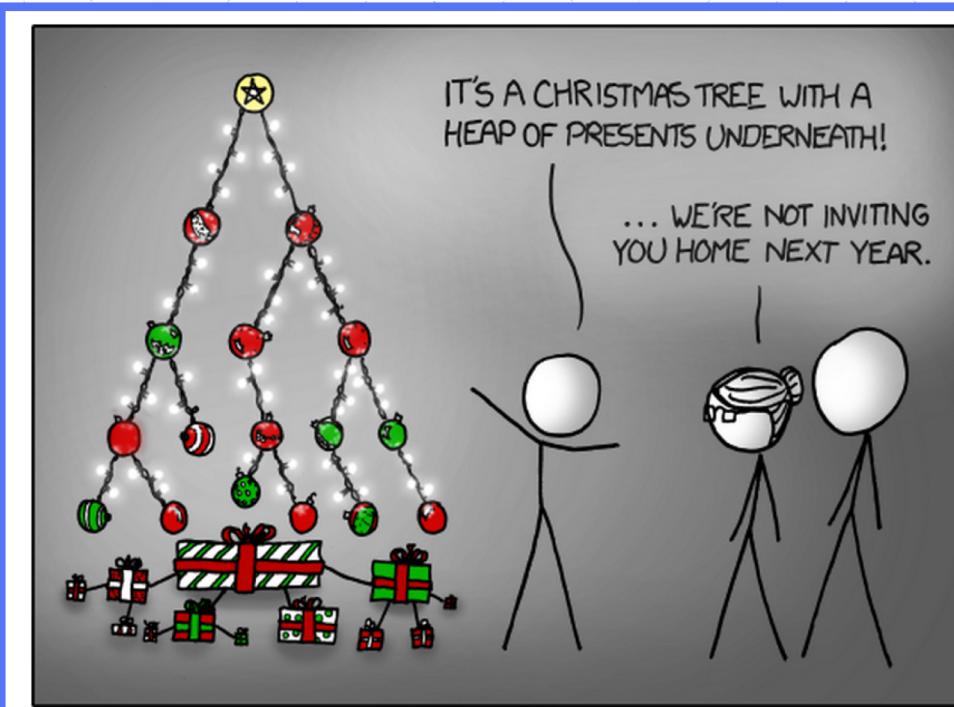


Presentación para usar con el libro de texto **Diseño y aplicaciones de algoritmos**, de MT Goodrich y R. Tamassia, Wiley, 2015

Montículo (Heap)



xkcd. <http://xkcd.com/835/>. "Tree." Used with permission under Creative Commons 2.5 License.

Operaciones de una Cola de Propiedades

- Una cola de prioridad almacena una colección de entradas.
- Cada entrada es un par (llave, valor)
- Principales métodos del ADT de cola prioritaria
 - **insertar** (k, v)
inserta una entrada con clave k y valor v
 - **removeMin ()**
elimina y devuelve la entrada con la llave más pequeña
- Métodos adicionales
 - **min()**
devuelve, pero no elimina, una entrada con la clave más pequeña
 - **tamaño(), está_vacio()**
- Aplicaciones:
 - Folletos en espera
 - Subastas
 - Motores de acciones

Ordenamiento con Colas de Propiedades



- Usamos una cola prioritaria.
 - Insertar los elementos con una serie de operaciones **de inserción**.
 - Elimine los elementos en orden con una serie de operaciones **removeMin**
- El tiempo de ejecución depende de la implementación de la cola de prioridad:
 - Una secuencia sin clasificar da tiempo de selección-ordenación: $O(n^2)$
 - Una secuencia ordenada proporciona ordenación por inserción: $O(n^2)$
- ¿Podemos hacerlo mejor?

Algorithm PQ-Sort(C, P):

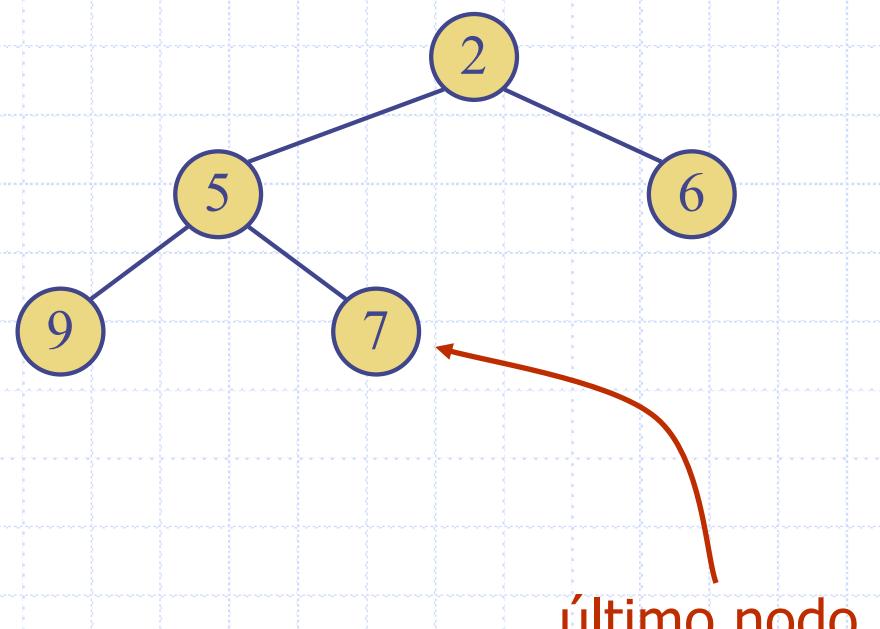
Input: An n -element array, C , index from 1 to n , and a priority queue P that compares keys, which are elements of C , using a total order relation

Output: The array C sorted by the total order relation

```
for  $i \leftarrow 1$  to  $n$  do
     $e \leftarrow C[i]$ 
     $P.insert(e, e)$       // the key is the element itself
for  $i \leftarrow 1$  to  $n$  do
     $e \leftarrow P.removeMin()$     // remove a smallest element from  $P$ 
     $C[i] \leftarrow e$ 
```

Montículos

- Un montículo es un árbol binario que almacena llaves en sus nodos y satisface las siguientes propiedades:
 - Orden del montículo: para cada nodo interno v que no sea la raíz, $\text{llave}(v) \geq \text{llave}(\text{padre}(v))$
 - Árbol binario completo: sea h la altura del montículo
 - para $i = 0, \dots, h - 1$, hay 2^i nodos de profundidad i
 - en profundidad $h - 1$, los nodos internos están a la izquierda de los nodos externos
- El **último nodo** de un montón es el nodo más a la derecha de máxima profundidad.





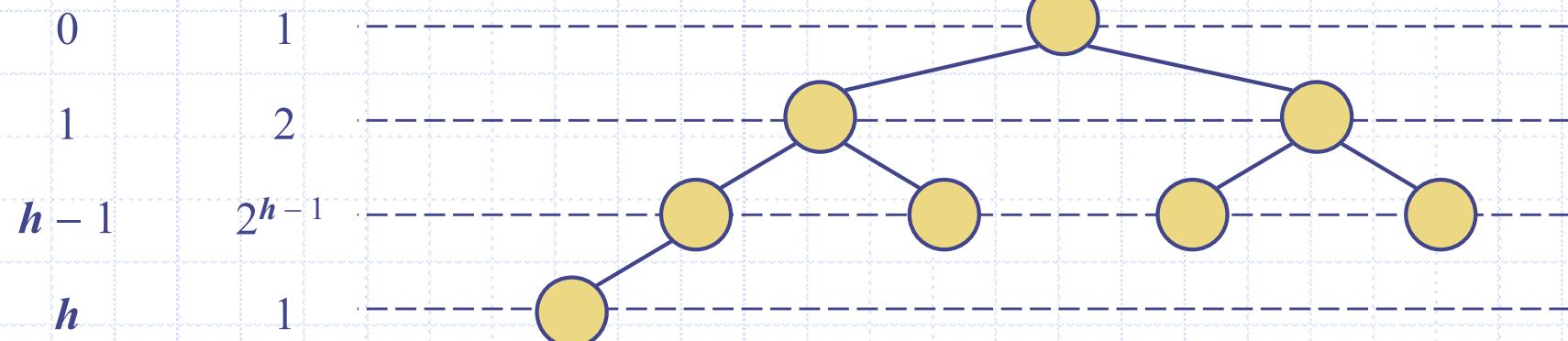
Altura de un Montículo

- **Teorema:** un montículo que almacena n las llaves tienen altura $O(\log n)$

Prueba: (aplicamos la propiedad del árbol binario completo)

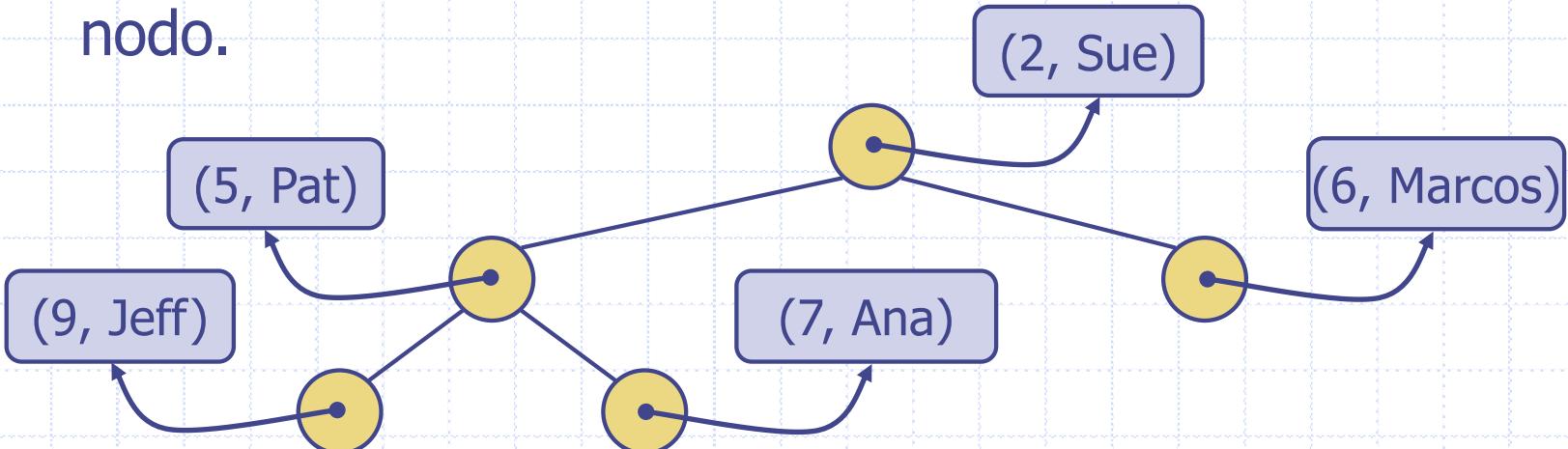
- Sea h la altura de un montículo que almacena n llaves
- Dado que hay 2^i llaves en la profundidad $i = 0, \dots, h - 1$ y al menos una llave en la profundidad h , tenemos $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
- Entonces, $n \geq 2^h$, es decir, $h \leq \log n$

profundidad llaves



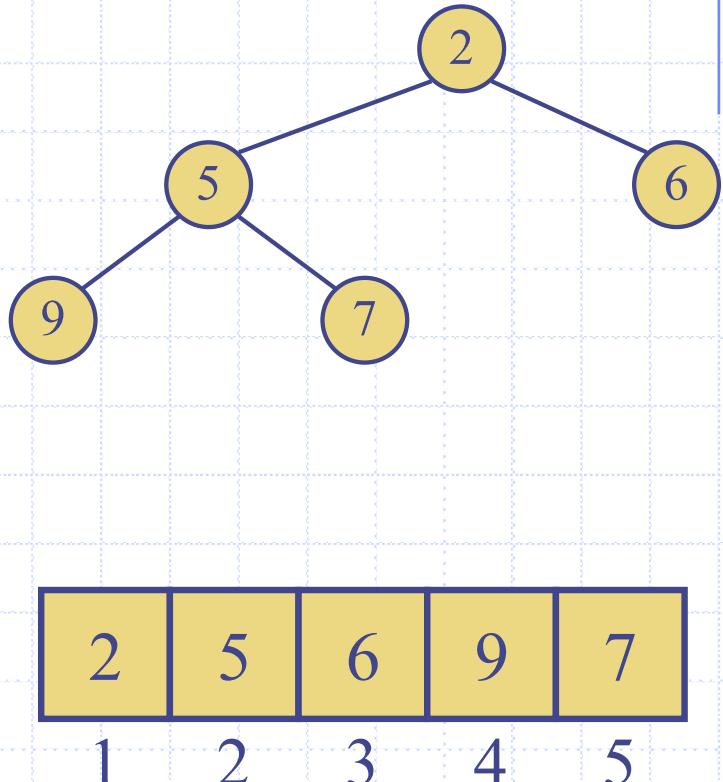
Montículos y Colas Prioritarias

- Podemos usar un montículo para implementar una cola prioritaria.
- Almacenamos un elemento (llave, elemento) en cada nodo interno.
- Realizamos un seguimiento de la posición del último nodo.



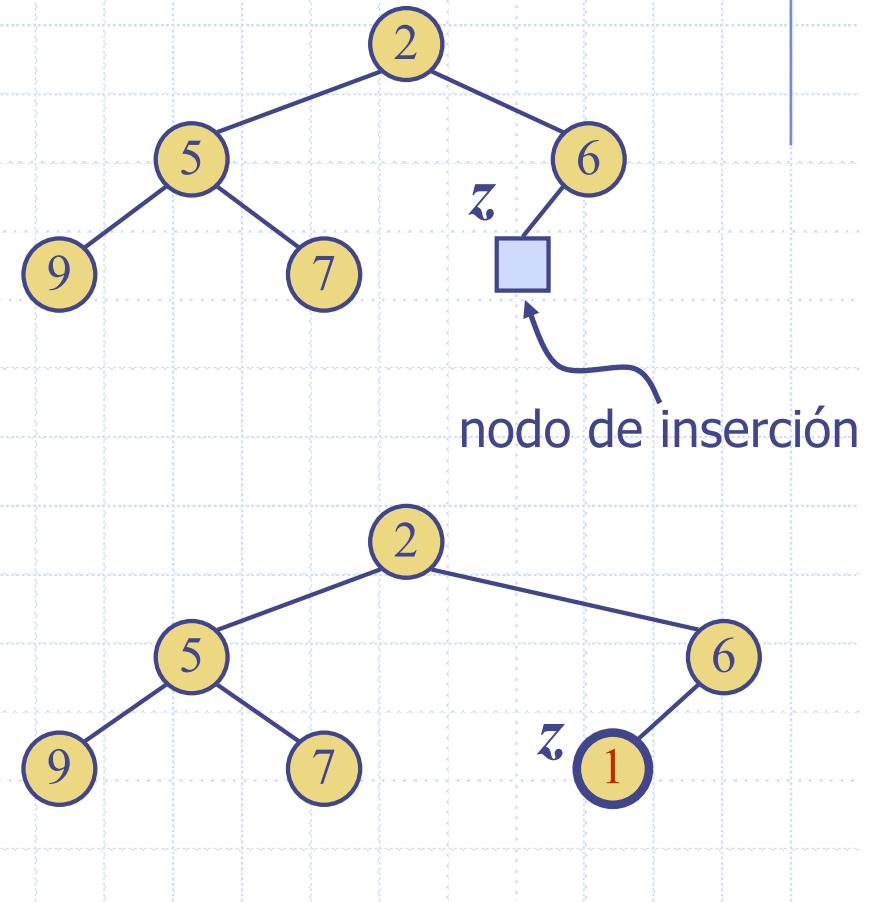
Implementación de Montículos basada en listas/arreglos

- Podemos representar un montículo con n llaves mediante un array de longitud n
- Para el nodo en el rango i
 - el hijo izquierdo está en el rango 2^i
 - el hijo derecho está en el rango 2^{i+1}
- Los enlaces entre nodos no se almacenan explícitamente
- La operación agregar corresponde a insertar en el rango $n + 1$
- La operación remove_min corresponde a eliminar en el rango n
- Produce clasificación de montículo in situ



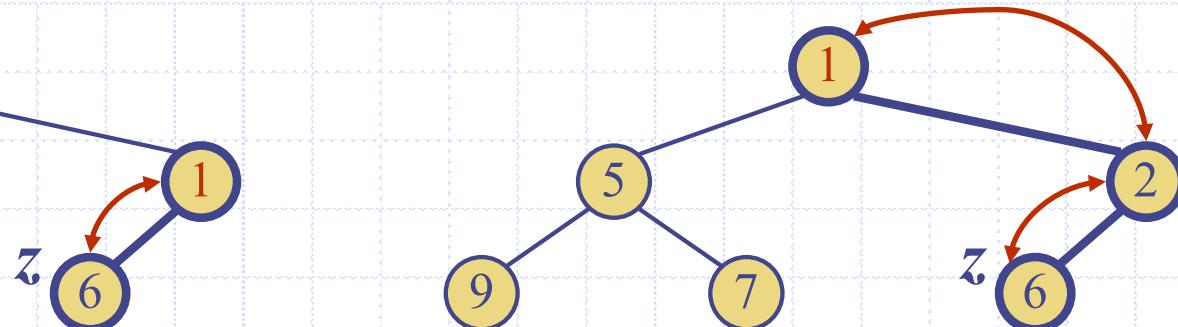
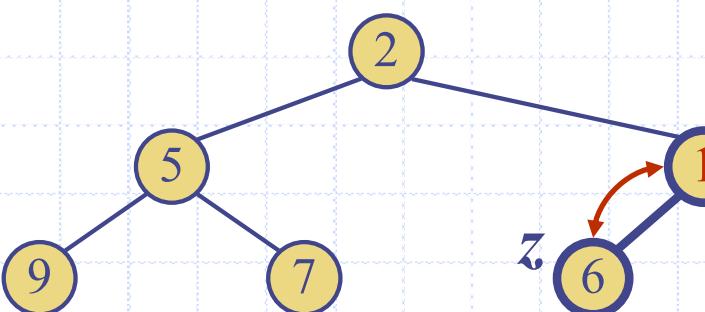
Inserción en un montículo

- El método `insertItem()` de la cola de prioridad corresponde a la inserción de una llave k en el montículo
- El algoritmo de inserción consta de tres pasos.
 - Encuentre el nodo de inserción z (el nuevo último nodo)
 - Almacenar k en z
 - Restaurar la propiedad de orden del montículo (que se analiza a continuación)



Inserción

- Después de la inserción de una nueva llave k , se puede violar la propiedad de orden del montículo
- El algoritmo inserción restaura la propiedad de orden del montículo intercambiando k a lo largo de una ruta ascendente desde el nodo de inserción
- La inserción termina cuando la llave k llega a la raíz o a un nodo cuyo padre tiene una llave menor o igual a k
- Dado que un montículo tiene una altura $O(\log n)$, el montículo se ejecuta en tiempo $O(\log n)$



Pseudocódigo de la Inserción

- Asumimos una implementación de montículo basada en listas/arreglos.

Algorithm HeapInsert(k, e):

Input: A key-element pair

Output: An update of the array, A , of n elements, for a heap, to add (k, e)

$n \leftarrow n + 1$

$A[n] \leftarrow (k, e)$

$i \leftarrow n$

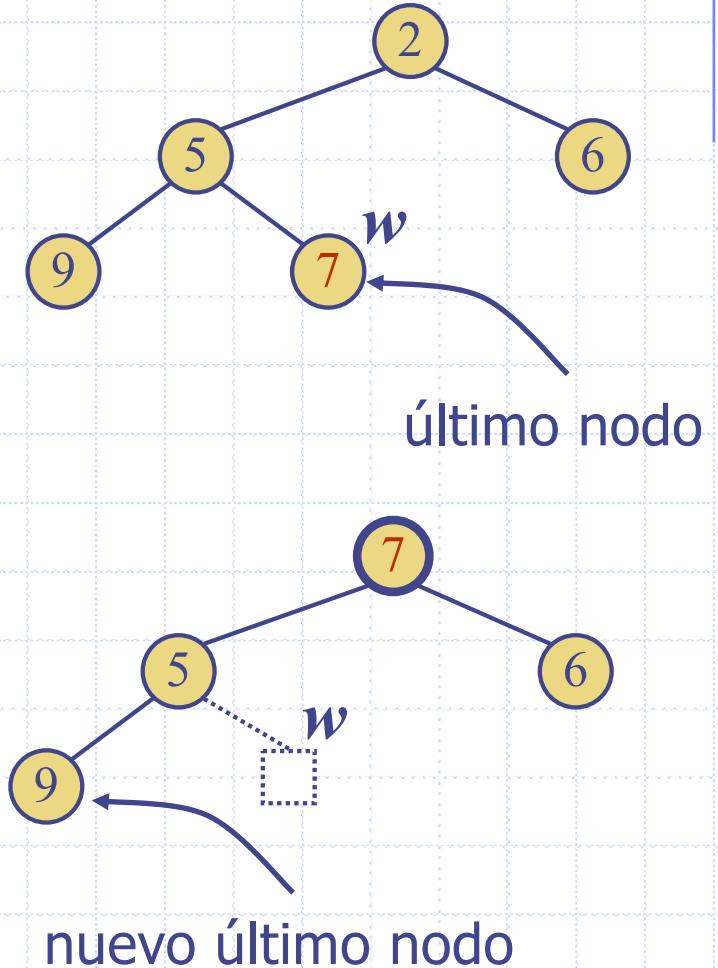
while $i > 1$ **and** $A[\lfloor i/2 \rfloor] > A[i]$ **do**

 Swap $A[\lfloor i/2 \rfloor]$ and $A[i]$

$i \leftarrow \lfloor i/2 \rfloor$

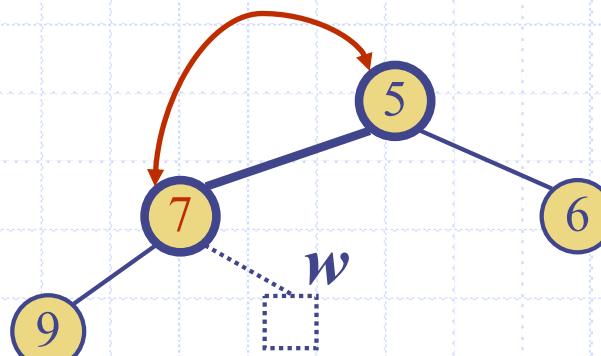
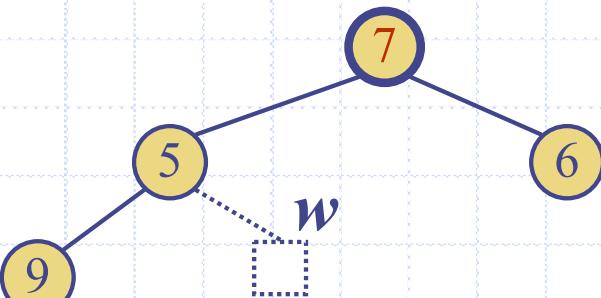
Eliminación de un Montículo

- El método `removeMin` de la cola de prioridad corresponde a la eliminación de la clave raíz del montículo
- El algoritmo de eliminación consta de tres pasos.
 - Reemplace la llave raíz con la clave del último nodo w
 - Quitar w
 - Restaurar la propiedad de orden del montón (que se analiza a continuación)



Eliminación

- Después de reemplazar la llave raíz con la llave k del último nodo, se puede violar la propiedad de orden del montículo
- El algoritmo de eliminación restaura la propiedad de orden del montículo intercambiando la llave k a lo largo de un camino descendente desde la raíz
- La inserción termina cuando la llave k llega a una hoja o un nodo cuyos hijos tienen claves mayores o iguales a k
- Dado que un montículo tiene una altura $O(\log n)$, la eliminación se ejecuta en tiempo $O(\log n)$



Eliminar Mínimo pseudocódigo

- Se supone que el montículo se implementa con una lista/arreglo

Algorithm HeapRemoveMin():

Input: None

Output: An update of the array, A , of n elements, for a heap, to remove and return an item with smallest key

$temp \leftarrow A[1]$

$A[1] \leftarrow A[n]$

$n \leftarrow n - 1$

$i \leftarrow 1$

while $i < n$ **do**

if $2i + 1 \leq n$ **then** // this node has two internal children

if $A[i] \leq A[2i]$ and $A[i] \leq A[2i + 1]$ **then**

return $temp$ // we have restored the heap-order property

else

 Let j be the index of the smaller of $A[2i]$ and $A[2i + 1]$

 Swap $A[i]$ and $A[j]$

$i \leftarrow j$

else // this node has zero or one internal child

if $2i \leq n$ **then** // this node has one internal child (the last node)

if $A[i] > A[2i]$ **then**

 Swap $A[i]$ and $A[2i]$

return $temp$ // we have restored the heap-order property

return $temp$ // we reached the last node or an external node

Performance de un montículo

- Un montículo tiene el siguiente rendimiento para las operaciones de cola prioritarias.

Operation	Time
insert	$O(\log n)$
removeMin	$O(\log n)$

- El análisis anterior se basa en los siguientes hechos:
 - La altura del montículo T es $O(\log n)$, ya que T está completo.
 - En el peor de los casos, la inserción y eliminación toma un tiempo proporcional a la altura de T.
 - Encontrar la posición de inserción en la ejecución de insertar y actualizar la posición del último nodo en la ejecución de removeMin requiere un tiempo constante.
 - El montículo T tiene n nodos internos, cada uno de los cuales almacena una referencia a una llave y una referencia a un elemento.

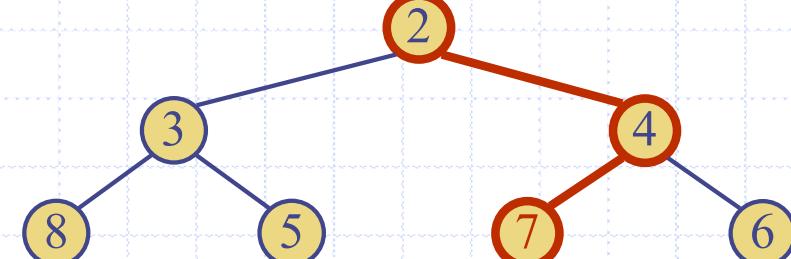
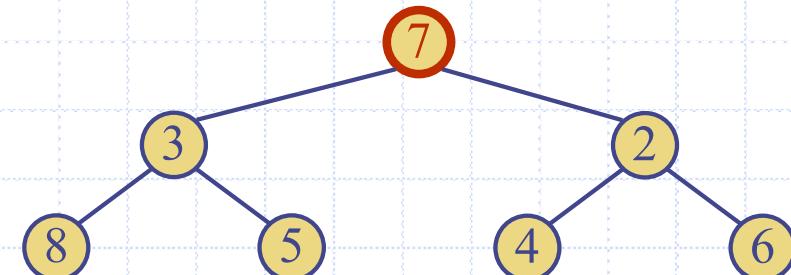
Ordenación con Montículos



- Considere una cola de prioridad con n elementos implementados mediante un montículo
 - el espacio utilizado es $O(n)$
 - métodos `insertar` y `eliminarMin` toma $O(\log n)$ tiempo
 - Los métodos `size`, `isEmpty` y `min` toman tiempo $O(1)$ tiempo.
- Usando una cola de prioridad basada en montículo, podemos ordenar una secuencia de n elementos en tiempo $O(n \log n)$
- El algoritmo resultante se llama Heap-Sort.
- Heap-Sort es mucho más rápido que los algoritmos de ordenación cuadrática, como la ordenación por inserción y la ordenación por selección.

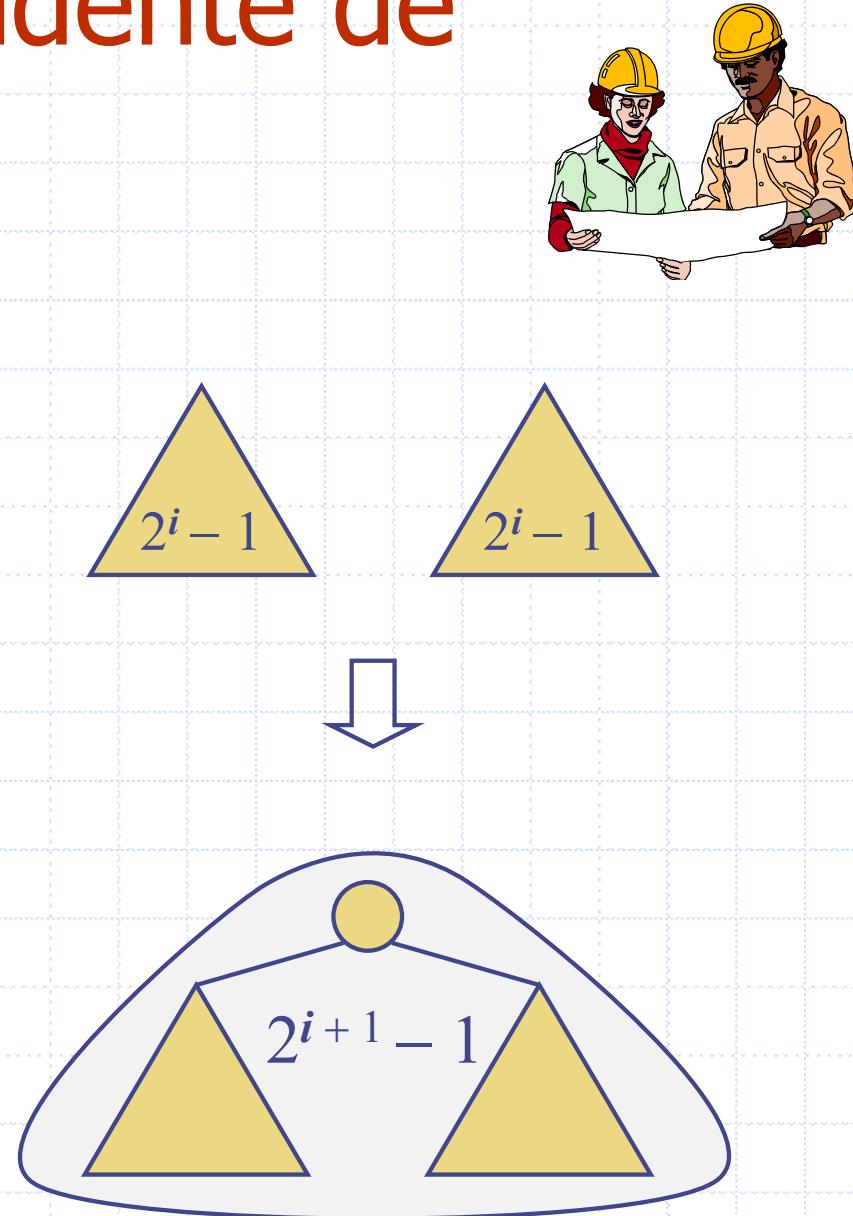
Fusionando dos montículos

- Nos dan dos dos montículos y una llave k .
- Creamos un nuevo montículo con el nodo raíz almacenando k y con los dos montículos como subárboles.
- Para restaurar la propiedad de orden del montículo aplicamos el algoritmo de inserción

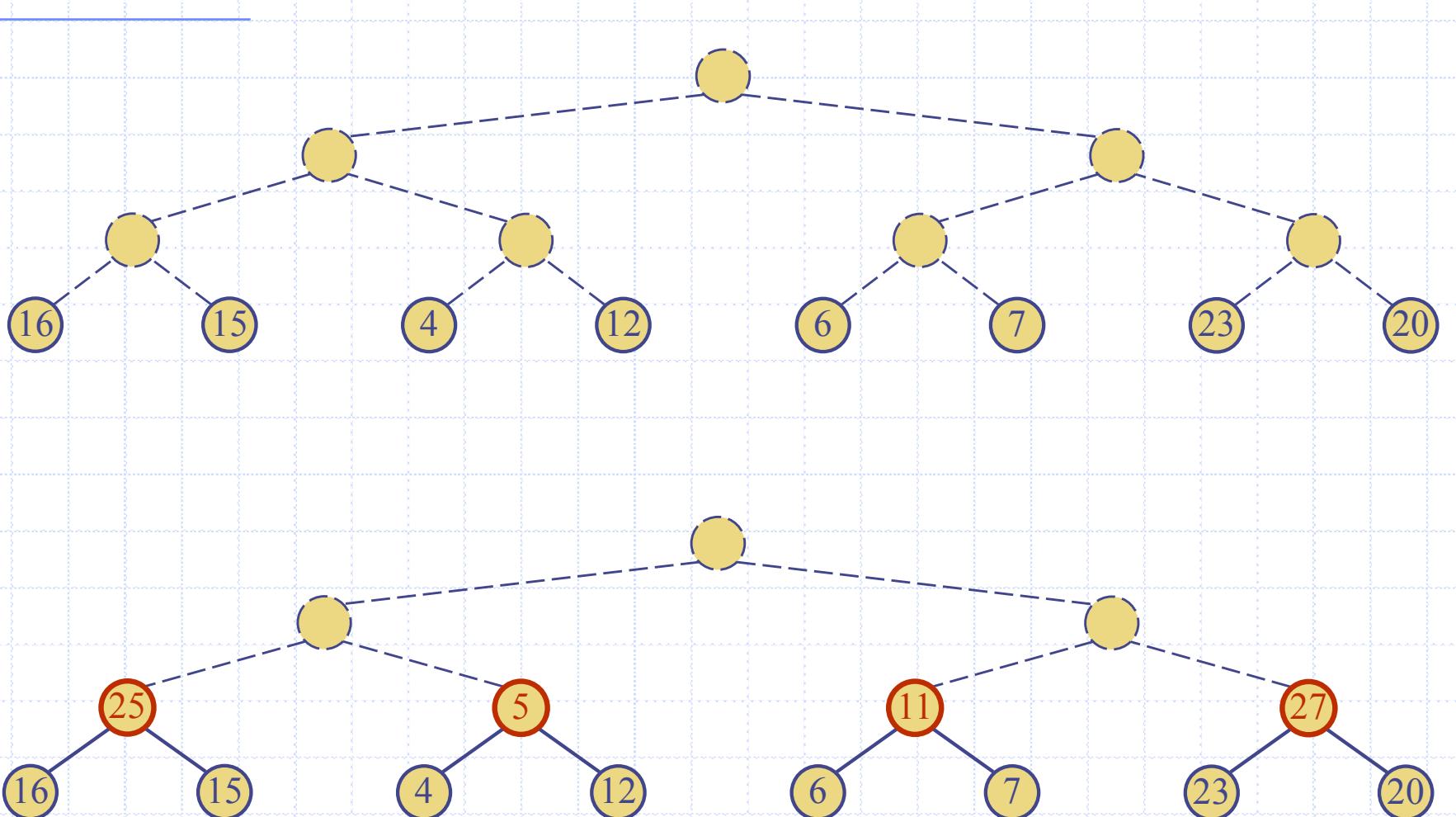


Construcción ascendente de un montículo

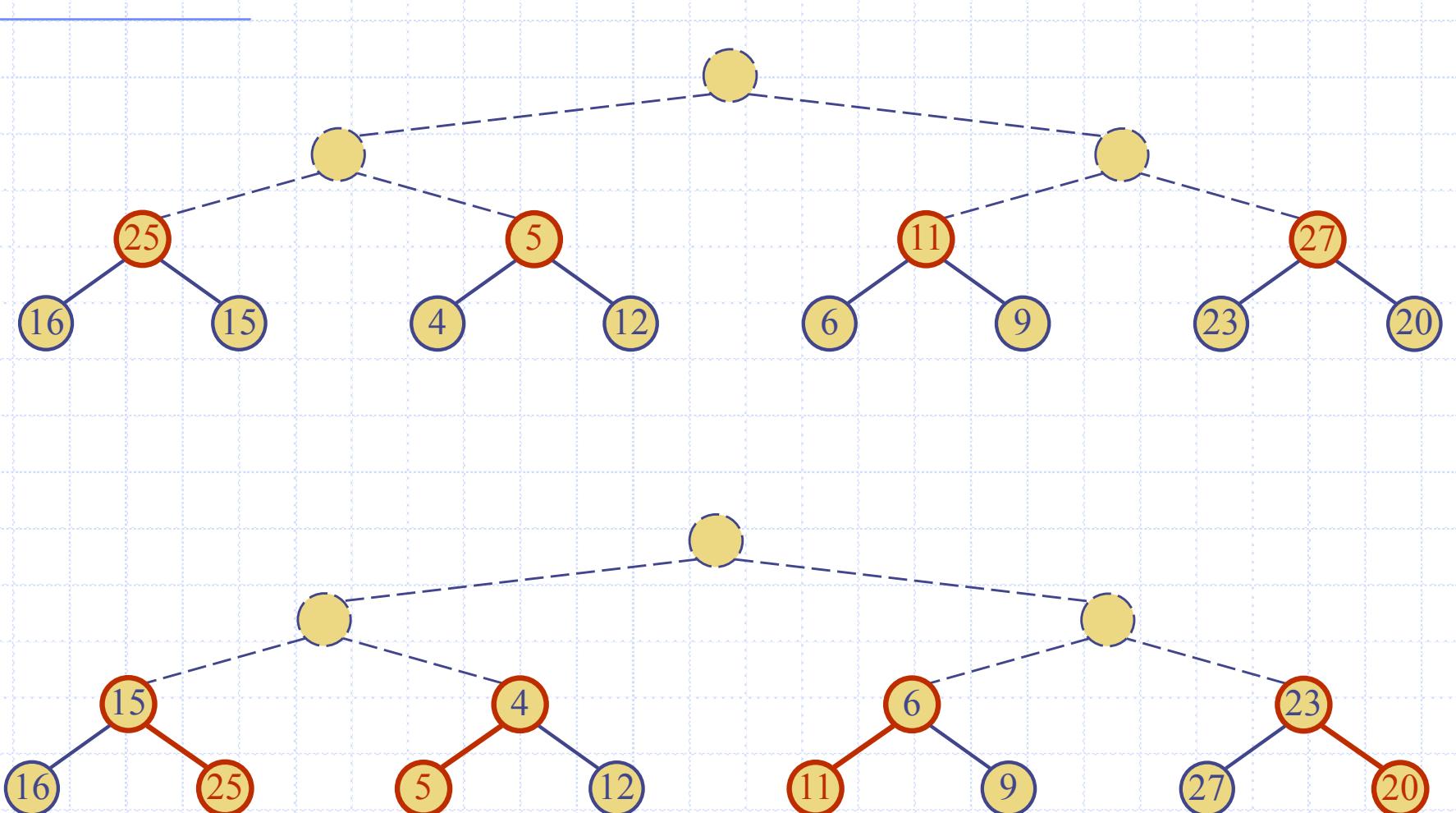
- Podemos construir un montículo que almacene n llaves dadas usando una construcción ascendente con $\log n$ fases
- En la fase i , los pares de montículos con $2^i - 1$ llaves se fusionan en montículos con $2^{i+1} - 1$ llaves



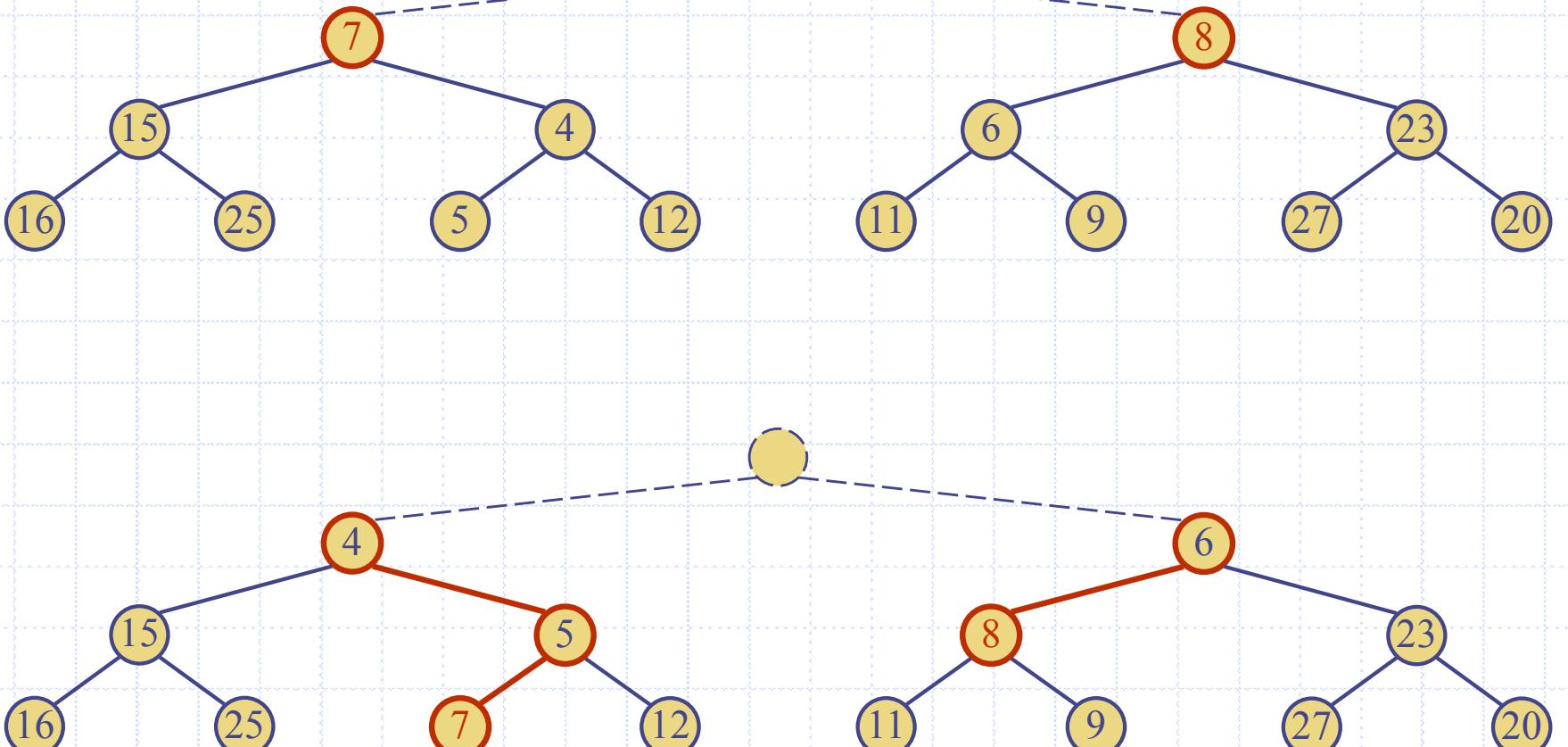
Ejemplo



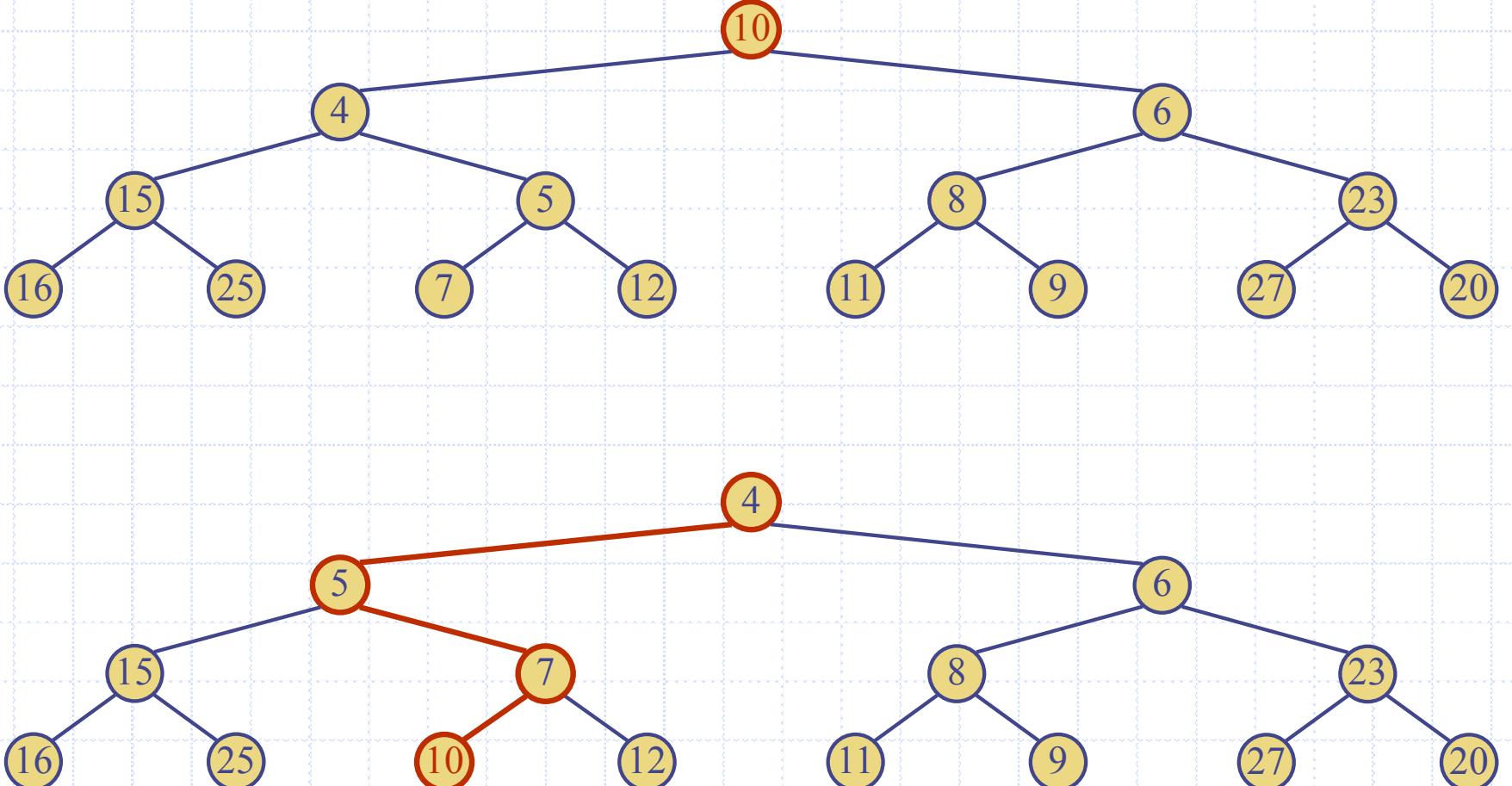
Ejemplo (cont.)



Ejemplo (cont.)



Ejemplo (fin)



Análisis

- Visualizamos el peor de los casos de un montículo con un camino substituto que va primero a la derecha y luego repetidamente a la izquierda hasta el final del montículo (este camino puede diferir del camino de inserción real)
- Dado que cada nodo es atravesado por como máximo dos caminos substitutos, el número total de nodos de los caminos substitutos es $O(n)$.
- Por lo tanto, la construcción ascendente del montículo se ejecuta en tiempo $O(n)$.
- La construcción ascendente del montículo es más rápida que n inserciones sucesivas y acelera la primera fase de la ordenación del montículo , que requiere tiempo $O(n \log n)$ en su segunda fase.

